Evaluation of AMD's Advanced Synchronization Facility Within a Complete Transactional Memory Stack

Dave Christie
Jae-Woong Chung
Stephan Diestelhorst
Michael Hohmuth
Martin Pohlack

Advanced Micro Devices, Inc.
ASF_Feedback@amd.com

Christof Fetzer
Martin Nowack
Torvald Riegel

Technische Universität Dresden
{*firstname.lastname*}@
inf.tu-dresden.de

Pascal Felber
Patrick Marlier
Etienne Rivière

Université de Neuchâtel
{*firstname.lastname*}@unine.ch

Abstract

AMD's Advanced Synchronization Facility (ASF) is an x86 instruction set extension proposal intended to simplify and speed up the synchronization of concurrent programs. In this paper, we report our experiences using ASF for implementing transactional memory. We have extended a C/C++ compiler to support language-level transactions and generate code that takes advantage of ASF. We use a software fallback mechanism for transactions that cannot be committed within ASF (e. g., because of hardware capacity limitations). Our evaluation uses a cycle-accurate x86 simulator that we have extended with ASF support. Building a complete ASF-based software stack allows us to evaluate the performance gains that a user-level program can obtain from ASF. Our measurements on a wide range of benchmarks indicate that the overheads traditionally associated with software transactional memories can be significantly reduced with the help of ASF.

Categories and Subject Descriptors C.1.4 [*Processor Architectures*]: Parallel Architectures; D.1.3 [*Programming Techniques*]: Concurrent Programming

General Terms Algorithms, Performance

Keywords Transactional Memory

1. Introduction

The number of cores per processor is expected to increase with each new processor generation. To take advantage of the processing capabilities of new multicore processors, applications must be able to harness the power of more and more cores. Amdahl's law [3] gives an upper bound on the application speedup s one can achieve: $s = \frac{1}{(1-P)+P/N}$ where P is the proportion of the program that can be made parallel and N is the number of cores. In other words, one can obtain a speedup that grows with the number of cores only if the fraction P can be made sufficiently large.

A promising way to increase the parallel fraction P is to use *speculation*. The idea is to execute blocks of code that could conflict (i. e., read or modify the same data) with blocks executed by other cores in such a way that (1) conflicts are detected dynamically and (2) state changes are only committed if it is guaranteed that there was no conflict. Speculation is an optimistic synchronization strategy that is especially helpful for improving the degree of parallelism in the following scenarios: first, when there is a good chance that two code blocks do not conflict because, for example, they access different memory locations; and second, when there is no easy way to predict at compile time if and when two code blocks will conflict, and pessimistic strategies like fine-grained locking unnecessarily limit scalability.

Transactional memory (TM) [15] is a shared-memory synchronization mechanism that supports speculation at the level of individual memory accesses. It allows to group any number of memory accesses into transactions, which are executed speculatively and take effect atomically only if there have been no conflicts with concurrect transactions executed by other threads. In the case of conflicts, transactions are rolled back and restarted. In programming languages, one can introduce *atomic block* constructs that are directly mapped onto transactions. Atomic blocks are also likely to be easier to use for programmers than other mechanisms such as fine-grained locking because they only specify what is required to be atomic but not how this is implemented.

While there are at least two industry implementations for hardware support for TM [8, 11], most of the current TM implementations are software-based [10, 14, 27]. Even the most efficient software TMs introduce significant overheads.

This lead some researchers to claim that software transactional memories (STMs) are only a research toy [7]. Our objective in this paper is to evaluate if a hardware extension recently proposed by AMD can help speed up concurrent applications that use speculation.

AMD's Advanced Synchronization Facility (ASF) is a public specification proposal of an instruction set extension for the AMD64 architecture [2]. It has the objective to reduce the overheads of speculation and simplify the programming of concurrent programs. ASF has been designed in such a way that it can be implemented in modern microprocessors with reasonable transistor budget and runtime overheads.

In this paper, we try to answer the following question: *can we use ASF to speed up the speculative execution of atomic blocks?* To that end, we implemented the ASF extensions in a near-cycle-accurate AMD64 simulator using ASF cycle costs and pipeline interactions that we would expect from a real hardware implementation. We also extended the software stack to work with ASF: we added support for atomic blocks to an existing open source C/C++ compiler and map the generated code onto the ASF primitives. If ASF cannot execute a block (e. g., because of capacity limitations), we use a software-based fallback solution.

Due to the lack of real applications with atomic blocks, we use a set of standard TM benchmarks in our evaluation. We compile these benchmarks with our extended C/C++ compiler into binaries that exploit ASF extensions for speculation. We show that ASF provides good scalability on several of the considered workloads while incurring much lower overhead than software-only implementations.

In the rest of this paper we continue with a description of the ASF specification and possible implementations (Section 2). Section 3 describes our TM software stack, including our TM compiler and our TM runtime for ASF. Section 4 presents our ASF simulator. A detailed evaluation of our hardware and software stack follows in Section 5. We discuss related work in Section 6 and conclude in Section 7.

2. Advanced Synchronization Facility (ASF)

ASF is an experimental AMD64 architecture extension proposal developed by AMD. Although ASF originally has been aimed at making lock-free programming significantly easier and faster, we were interested in applying ASF to transactional programming, especially to accelerating TM systems.

2.1 ASF rationale

ASF is purely experimental and has not been announced for any future product. However, it has been developed in the framework of constraints that apply to the development of a high-volume microprocessor. In general, academia has little insight into how constrained the opportunities to innovate in this environment are. Thus, we think that one contribution of ASF is that it helps setting expectations on what can possibly be anticipated in future products.

Today's commercial processors are very complex (they contain billions of transistors) and require a large design and verification effort. Market pressures impose the need to be functional and on time. Therefore, these processors typically cannot serve as a vehicle for experimentation. Before any complex new feature can be added to a product, a demonstration of broad benefits is required. Additionally, new ground-up processor designs are increasingly rare because they are extremely expensive. Typically, new processor generations are instead incremental evolutions of older processor designs.

These constraints had implications on ASF's design that resulted in differences from many academic hardware-extension proposals. We refrain from mandating modifications to critical components such as the cache-coherence protocol, and instead allow leveraging existing processor components as much as possible: caches and store buffers may be used for data monitoring and versioning, and the hardware's contention management piggybacks on the existing cache-coherence protocol. It follows that cache lines are the units of protection, and that only very simple contention management can be implemented in hardware: ASF uses a straightforward requester-wins scheme, which always aborts the transaction already containing the conflicting element in its working set. Without changes to the cache-coherence protocol, the cores retain their existing system interface. This makes ASF trivially available for larger cache-coherent multiprocessor systems, and not only for single-chip multiprocessors.

Another implication of the design constraints is that ASF has a detailed specification, which we developed for one main reason: we wanted to ensure that ASF can be implemented in various ways (without constraining implementation freedom too much), so we needed to avoid the pitfall of using implementation artifacts as architecture. The only way of doing this is documenting all corner cases and defining a sufficiently general behavior. Examples of potential architecture holes that need to be closed are ASF's behavior under virtualization or debugging, and ASF's interaction with the paging hardware.

Nonetheless, ASF provides several features that existing microarchitectures can accommodate with relative ease: architecturally ensured forward progress up to a certain transaction capacity, and a mechanism for selectively annotating memory accesses as either transactional or nontransactional.

2.2 ASF specification

ASF provides seven new instructions for entering and leaving speculative code regions (*speculative regions* for short), and for accessing protected memory locations (i. e., memory locations that can be read and written speculatively and which abort the speculative region if accessed by another thread): `SPECULATE`, `COMMIT`, `ABORT`, `LOCK MOV`, `WATCHR`, `WATCHW`, and `RELEASE`. All of these instructions are available in all system modes (user, kernel; virtual-machine guest,

```
; DCAS Operation:
; IF ((mem1 = RAX) && (mem2 = RBX)) {
;   mem1 = RDI;      mem2 = RSI;     RCX = 0;
; } ELSE {
;   RAX = mem1;      RBX = mem2;     RCX = 1;
; }  // (R8, R9, R10 modified)
DCAS:
  MOV      R8, RAX
  MOV      R9, RBX
retry:
  SPECULATE             ; Speculative region begins
  JNZ      retry        ; Page fault, interrupt, or contention
  MOV      RCX, 1       ; Default result, overwritten on success
  LOCK MOV R10, [mem1]  ; Specification begins
  LOCK MOV RBX, [mem2]
  CMP      R8, R10      ; DCAS semantics
  JNZ      out
  CMP      R9, RBX
  JNZ      out
  LOCK MOV [mem1], RDI  ; Update protected memory
  LOCK MOV [mem2], RSI
  XOR      RCX, RCX     ; Success indication
out:
  COMMIT
  MOV      RAX, R10
```

Figure 1. ASF example: An implementation of a DCAS primitive using ASF.

host). Figure 1 shows an example of a double CAS (DCAS) primitive implemented using ASF.

Speculative-region structure. Speculative regions have the following structure. The SPECULATE instruction signifies the start of such a region. It also defines the point to which control is passed if the speculative region aborts: in this case, execution continues at the instruction following the SPECULATE instruction (with an error code in the rAX register and the zero flag cleared, allowing subsequent code to branch to an abort handler).

The code in the speculative region indicates protected memory locations using the LOCK MOV, WATCHR, and WATCHW instructions. The first is also used to load and store protected data; the latter two merely start monitoring a memory line for concurrent stores (WATCHR) or loads and stores (WATCHW).

Speculative regions can optionally use the RELEASE instruction to modify a transaction's read set. With RELEASE, it is possible to stop monitoring a read-only memory line, but not to cancel a pending transactional store (the latter is possible only with ABORT). RELEASE, which is strictly a hint to the CPU, helps decrease the odds of overflowing transactional capacity and is useful, for example, when walking a linked list to find an element that needs to be mutated.

COMMIT and ABORT signify the end of a speculative region. COMMIT makes all speculative modifications instantly visible to all other CPUs, whereas ABORT discards these modifications.

ASF supports dynamic nesting, which allows simple composition of multiple speculative regions into an overarching speculative region up to a maximum nesting depth (256). Nesting is implemented by flattening the hierarchy of speculative regions (*flat nesting*): memory locations protected by a nested speculative region remain protected until the outermost speculative region ends, and aborts inside a nested speculative region cause rollback of the whole outermost speculative region.

Aborts. Besides the ABORT instruction, there are several conditions that can lead to the abort of a speculative region: contention for protected memory; system calls, exceptions, and interrupts; the use of certain disallowed instructions; and, implementation-specific transient conditions. Unlike in Sun's hardware transactional memory (HTM) design [11], TLB misses do not cause an abort.

In case of an abort, all modifications to protected memory locations are undone, and execution flow is rolled back to the beginning of the speculative region by resetting the instruction and stack pointers to the values they had directly after the SPECULATE instruction. No other register is rolled back; software is responsible for saving and restoring any context that is needed in the abort handler. Additionally, the reason for the abort is passed in the rAX register.

Because all privilege-level switches (including interrupts) abort speculative regions and no ASF state is preserved across such a context switch, all system components (user programs, OS kernel, hypervisor) can make use of ASF without interfering with one another. This differs from Azul Systems' HTM design [8], which appears to maintain transactions across system calls.

Selective annotation. Unlike most other architecture extensions aimed at the acceleration of transactions, ASF allows software to use both transactional and nontransactional memory accesses within a speculative region.[1] This feature allows reducing the pressure on hardware resources providing TM capacity because programs can avoid protecting data that is known to be thread-local. It also allows implementing STM runtimes or debugging facilities (such as shared event counters) that access memory directly without risking aborts because of memory contention.

For example, our compiler (described in in Section 3.1) automatically makes use of selective annotation to avoid protecting the local thread's stack whenever possible.

Because ASF uses cache-line-sized memory blocks as its unit of protection, software must take care to avoid colocating both protected and unprotected memory objects in the same cache line. ASF can deal with some colocation scenarios by hoisting colocated objects accessed using unprotected memory accesses into the transactional data set. However, ASF does not allow unprotected writes to memory lines that have been modified speculatively and raises an exception if that happens.

Isolation. ASF provides strong isolation: it protects speculative regions against conflicting memory accesses to protected memory locations from both other speculative regions and regular code concurrently running on other CPUs.

[1] Each MOV instruction can be selectively annotated to be either transactional (with LOCK prefix) or nontransactional (no prefix); hence the name *selective annotation*.

In addition, all aborts caused by contention appear to be instantaneous: ASF does not allow any side effects caused by misspeculation in a speculative region to become visible. These side effects include nonspeculative memory modifications and page faults after the abort, which may have been rendered spurious or invalid by the memory access causing the abort.

Eventual forward progress. ASF architecturally ensures eventual forward progress in the absence of contention and exceptions when a speculative region protects not more than four 64-byte memory lines.[2] This enables easy lock-free programming without requiring software to provide a second code path that does not use ASF. Because it only holds in the absence of contention, software still has to control contention to avoid livelock, but that can be accomplished easily, for example, by employing an exponential-backoff scheme.

An ASF implementation may have a much higher capacity than the four architectural memory lines, but software cannot rely on any forward progress if it attempts to use more than four lines. In this case, software has to provide a fallback path to be taken in the event of a capacity overflow, for example, by grabbing a global lock monitored by all other speculative regions.

2.3 ASF implementation variants

We designed ASF such that a CPU design can implement ASF in various ways. The minimal capacity requirements for an ASF implementation (four transactional cache lines) are deliberately low so existing CPU designs can support simple ASF applications, such as lock-free algorithms or small transactions, with very low additional cost. On the other side of the implementation spectrum, an ASF implementation can support even large transactions efficiently.

In this section, we present two basic implementation variants and a third, hybrid, variant. We implemented two of these three variants in the simulator we used in our evaluation (described in Section 4).

Cache-based implementation. A first variant is to keep the transactional data in each CPU core's L1 cache and to use the regular cache-coherence protocol for monitoring the transactional data set.

Each cache line needs two additional bits: a speculative-read and a speculative-write bit.[3] When a speculative region protects data cached in a given line, the speculative-read bit is turned on. Whenever a cache line that has this bit set needs to be removed from the cache (because of a remote write request or because of a capacity conflict), the speculative region is aborted.

The speculative-write bit is set in addition to the speculative-read bit when a speculative region modifies protected data (or uses `WATCHW` to protect it). When this happens to a dirty cache line, the L1 cache must first write back the modified data to a backup location (to main memory or to a higher-level cache).

When a speculative region completes successfully, all speculative-read and speculative-write bits are flash-cleared. In this case, the current values in L1 become authoritative and visible to the remainder of the system. On the other hand, if a speculative region is aborted, the cache must first invalidate all cache lines that have the speculative-write bit set before clearing the speculative-read and speculative-write bits.

This implementation has the advantage that, potentially, the complete L1 cache capacity is at disposal for transactional data. However, the capacity is limited by the cache's associativity. Additionally, an implementation that wants to provide the (associativity-independent) architectural minimum capacity of four memory lines using the L1 needs to ensure that each cache index can hold at least four cache transactional lines that cannot be evicted by nontransactional data refills.

LLB-based implementation. Another ASF implementation variant is to introduce a new CPU data structure called the *locked-line buffer* (LLB). The LLB holds the addresses of protected memory locations as well as backup copies of speculatively modified memory lines. It snoops remote memory requests, and if an incompatible probe request is received, it aborts the speculative region and writes back the backup copies before the probe is answered.

The advantage of an LLB-based implementation is that the cache hierarchy does not have to be modified. Speculatively modified cache lines can even be evicted to another cache level or to main memory.[4]

Because the LLB is a fully associative structure, it is not bound by the L1 cache's associativity and can ensure a larger number of protected memory locations. However, since fully associative structures are more costly, the total capacity typically would be much smaller than the L1 size.

Hybrid implementation. It is also possible to combine aspects of a cache-based and an LLB-based implementation. We propose using the L1 cache to monitor the speculative region's read set, and the LLB to maintain backup copies of and monitor its write set.

In this design, each L1 cache line needs only one speculative-read bit. The LLB makes the speculative-write bit redundant. When the speculative region modifies a protected cache line, the backup data is copied to the LLB. Thus, dirty

[2] *Eventual* means that there may be transient conditions that lead to spurious aborts, but eventually the speculative region will succeed when retried continuously. The expectation is that spurious aborts almost never occur and speculative regions succeed the first time in the vast majority of cases.

[3] We assume that the L1 cache is not shared by more than one logical CPU (hardware thread).

[4] We assume the LLB can snoop probes independently from the caches and is not affected by cache-line evictions.

cache lines do not have to be backed up by evicting them to a higher cache level or main memory.

In comparison to a pure cache-based implementation, this design minimizes changes to the cache hierarchy, especially when the all caches participate in the coherence protocols as first class citizens: the CPU core's L1 cache remains the owner of the cache line and can defer responses to incompatible memory probes until it has written back the backup data, without having to synchronize with other caches.

The advantage over a pure LLB-based implementation is the much higher read-set capacity offered by the L1 cache.

3. Integrating ASF with transactional C/C++

Recently, there have been proposals to add transactional language constructs to C and C++ [17], by allowing programmers to specify atomic blocks within programs. We have built a compiler, the Dresden TM Compiler (DTMC), and a runtime library, ASF-TM, to execute such transactional C and C++ programs with the help of ASF. To evaluate ASF, we use a software stack that spans from language-level atomic blocks (called transaction statements in the proposals) down to ASF hardware. This permits us to measure more accurately how much benefit applications might gain from using ASF. In particular, we are able to measure potential overheads that are introduced when translating atomic blocks into ASF transactions.

Because much of concurrent software written in C and C++ is based on locks and not atomic blocks, our software stack also supports existing software with the help of lock elision [25]. In this paper, we only focus on the evaluation of programs that use atomic blocks.

We first present our compiler, DTMC, in Section 3.1, explain the design of a runtime library that uses ASF in Section 3.2, discuss how to safely execute nonspeculative code in Section 3.3, and summarize lessons learned in Section 3.4.

3.1 Dresden TM Compiler

A compiler supporting atomic blocks has to transform language-level transactions into machine code that ensures the atomic executions of the blocks. This could be done with the help of software transactional memory, locks, or—like in our case—with the help of ASF.

Atomic blocks come in the form of a C/C++ statement that a programmer can use to declare that a block of code should be executed as a transaction. Our compiler supports a large subset of the transactional language constructs for C++ that have recently been proposed by engineers from Intel, IBM, and Sun [17].[5]

DTMC transforms transactional C/C++ programs in a multipass process. It is based on the LLVM compiler framework [20], which allows the construction of highly modular compilers. LLVM's compiler front-end for C/C++ (`llvm-gcc`) parses and transforms source code into LLVM's intermediate representation (IR). To support transaction statements, we took the TM support code that Red Hat engineers are developing for the GNU Compiler Collection (`gcc-tm`) and ported it to `llvm-gcc`. The output of our modified `llvm-gcc` is thus LLVM IR in which transaction statements are visible.

DTMC maps the transaction statements of the LLVM IR to calls to a TM runtime library. It uses a compiler pass that transforms LLVM IR with transaction statements so (1) memory accesses in transactions are rewritten as calls to load and store functions in the TM runtime library, (2) transactions are started and committed using calls to the TM library, and (3) function calls inside transactions are redirected to "transactional" clones of the original functions. This compiler pass is a much improved and extended version of Tanger [13].

The application binary interface (ABI) of the TM runtime library follows a proposal by Intel [18]. This ABI is not ASF-specific, but rather designed to be compatible with many existing STM algorithms and implementations. We want our compiler to target libraries that provide this ABI instead of generating ASF code directly because this makes the TM compiler independent of the TM implementation, and it also allows linking the TM implementation to the application either statically or dynamically.

The use of an intermediate TM library will permit running the same binary code on machines regardless of whether they support ASF or not. Moreover, the use of a standardized ABI permits programmers to mix compilers and STM implementations from different vendors. In particular, applications can already be developed using the ABI and an STM implementation. When new hardware features like ASF would become widely available, the applications could use them without recompilation.

Having an intermediate TM library that uses an ABI not specifically designed for ASF can introduce run-time overheads. Our approach is to use link-time optimization to reduce or even eliminate these overheads. In LLVM, the intermediate representation of the code is still available at the final linking stage when creating the application's executable code. This allows the compiler to perform whole-program optimization and code generation, which includes inlining the functions in the TM library if this library is linked statically. This generally results in code of the same quality as if the compiler inserted the TM instrumentation code directly. Our compiler can also create different code paths for a transaction. These code paths use functions for different runtime modes of the TM, and the TM library determines at runtime (i. e., when starting or restarting a transaction) which code path will be executed. For example, an STM and an ASF code path can coexist and can be optimized independently. We used static linking and link-time optimization when creating the application code evaluated in Section 5.

[5] It supports transaction statements but does not yet support TM-specific attributes for functions and classes.

```
extern long cntr;
void increment() {
  __tm_atomic {

    cntr = cntr + 5;



  }
}
```

```
extern long cntr;
void increment() {
  _ITM_beginTransaction(...);

  long l_cntr = (long) _ITM_R8(&cntr);
  l_cntr = l_cntr + 5;
  _ITM_W8(&cntr, l_cntr);
  _ITM_commitTransaction();
}
```

```
; mem1 for cntr

SPECULATE
JNZ      handle_abort
LOCK MOV RCX, [mem1]
ADD      RCX, 5
LOCK MOV [mem1], RCX
COMMIT
```

Figure 2. An example of how C code with a transaction statement (left) is transformed to targeting a TM library ABI (middle) and finally to native code that uses ASF (right). Note that additional code around `SPECULATE` for providing full semantics of `_ITM_beginTransaction` has been omitted for brevity.

Figure 2 shows the transformation stages of a simple atomic block in C code. There is no dependence on ASF before ASF-TM is linked to the application (last transformation stage), still link-time optimization can inline ASF instructions. Please note that several implementation details have been omitted for clarity (e. g., ASF-TM requires more software support to begin and commit transactions).

3.2 ASF-TM

ASF-TM is our TM library that implements the TM ABI using ASF. It adds (1) some features that are required by the ABI but are not part of ASF and (2) a fallback execution path in software. We need a fallback path in case ASF cannot commit a transaction because of one of ASF's limitations (e. g., capacity limitations or a transaction executing a system call; see Section 2).

We chose to just provide a serial-irrevocable mode as the software fallback. This mode already exists in most STMs as the fallback path for execution of external, nonisolated, or irrevocable actions. It is also required by the ABI. If a transaction is in this mode, it is not allowed to abort itself, but the TM ensures that it will not be aborted and that no other transaction is being executed concurrently. If no transaction is in this mode, all transactions execute the normal TM algorithm (in our case, ASF speculative regions). Our measurement shows that ASF can handle most of our current workloads directly in hardware (see Section 5).

If a more elaborate fallback mechanism is needed in a later version of ASF-TM, one could switch between STM or ASF transactions (similar to PhasedTM [21]), or one could ensure that STM transactions can safely run concurrently with ASF transactions (similar to Hybrid TM [9]).

ASF-TM needs to make sure that conflicting accesses by concurrent transactions are detected. To do so, it uses ASF speculative loads and stores for these accesses. This is implemented using ASF assembly code in ASF-TM. Note that this code will get inlined if we link ASF-TM statically to the application. Our compiler only uses transactional memory accesses for data that is potentially shared with other threads. Therefore, accesses to a thread's stack are not speculative or transactional unless the address of a variable on the stack has been taken.

As we explained previously, we want ASF-TM to be compatible with the existing TM ABI, so we cannot rely on the compiler to insert a `SPECULATE` instruction into the application code. Instead, transactions are started by calling a special "transaction begin" function that is a combination of a software `setjmp` implementation and a `SPECULATE` instruction. Because ASF does not restore CPU registers (except the instruction and stack pointers), we use the software `setjmp` to checkpoint and restore CPU registers[6] in the current thread's transaction descriptor. Additionally, we partially save the call stack to allow the function to return a second time in the event of an abort.

When ASF detects a conflict, it aborts by rolling back all speculatively modified cache lines and resuming execution at the instruction that follows the `SPECULATE` instruction, which is located in the "transaction begin" function. Transaction restarts are then emulated by letting the application return from this function again, thus making it seem as if the previous attempt at running the transaction never happened. The function returns a (TM-ABI-defined) value that indicates whether changes to the stack that have not been tracked by ASF have to be rolled back, and which code path (e. g., ASF or serial-irrevocable mode) has to be executed. Our compiler adds code that performs the necessary actions according to the return value.

Before starting the ASF speculative region, the begin function additionally initializes the tracking of memory management functions (see Section 3.3) and performs simple contention management if necessary (e. g., use exponential back-off). ASF transactions that fail to execute a certain number of times or experience ASF capacity overflows will get restarted in serial-irrevocable mode.

To commit an ASF transaction, it is sufficient to call a commit function of the ASF-TM library that contains an ASF `COMMIT` instruction.

3.3 Safely executing nonspeculative code

There are a few challenges when implementing ASF-TM. ASF permits nonspeculative memory accesses within transactions. This allows the reduction of the read-set size of a transaction and, hence, larger transactions can be executed with ASF. However, as a consequence, we need to take care of nonspeculative code called from within an ASF speculative region.

[6] The calling convention that is used in the application code determines which registers have to be restored.

ASF requires programmers or compilers to explicitly mark memory accesses within a transaction that are speculative. If a nontransactional function f (e. g., within an external library) were to be called within an ASF speculative region, all memory accesses of this function would be nonspeculative. These nonspeculative memory updates of f could cause consistency issues if the region is aborted.

Transactions might call external functions for several reasons: for example, memory management or exception handling. STMs therefore deal with calls to external functions in different ways: (1) by providing a transactional version of the function in the TM library or by the programmer; (2) by relying on the programmer to mark functions that can safely be executed from within a software transaction; or, (3) by falling back to serial-irrevocable mode.

ASF-TM uses Approach 1, for example, for a transactional `malloc` function. Because the semantics of this function are known, the transactional version can be built so it is robust against asynchronous aborts by ASF. This is particularly easy to ensure for functions that only operate on thread-local data. For example, ASF-TM uses a custom memory allocator for in-transaction allocations to avoid having to abort and execute in serial-irrevocable mode. This allocator still uses the default allocator internally, but executing the standard `malloc` function in a speculative region would not be safe because of potential incomplete updates to the memory allocation metadata.

ASF-TM can, but currently does not, support Approach 2. The problem with this approach is that nontransactional functions can be aborted at any point when using ASF (e. g., if a memory location that has been speculatively read is modified in another thread). Such asynchronous aborts are not possible in STM-based systems [18], and it is easier for a programmer to determine if it is safe to call a nontransactional function within a transaction in such systems. Hence, ASF-TM's safety requirements are different than those of STMs, and it is not clear that expecting the programmer to consider both is beneficial in the long term.

When compiling for ASF-TM, DTMC will always use Approach 3 (i. e., switch a transaction to serial-irrevocable mode) before calling a function for which there exists no ASF-safe version.

3.4 Lessons learned

ASF is very well aligned with a standard software stack. From our integration of ASF with ASF-TM and DTMC, we learned that the current ASF signaling mechanisms [2] could be improved to reduce the overhead and the complexity of ASF-TM. In particular, reporting errors via the `SPECULATE` instruction—instead of by generating exceptions—simplified ASF-TM. We therefore chose to implement a variant of ASF in the ASF simulator and built our software for this variant. We hope that a future revision of the ASF specification will reflect these changes.

4. ASF simulator

For our evaluation of ASF, we have to rely largely on simulation, because the costs of implementing and verifying the implementation of such a large-scale feature in a commercial microprocessor are prohibitive and hard to justify while still exploring the design space.

PTLsim [32] has been chosen out of the wealth of available simulators since it initially fulfilled many of our requirements:

- **AMD64 ISA simulation:** ASF is specified as an extension to AMD's established AMD64 instruction set architecture (ISA); therefore, it is crucial for the simulator to support the same ISA. In addition, this support allows us to easily reuse the existing compiler infrastructure, binaries, and compiled operating system kernels. Using the same binary code will generate more relevant performance predictions and comparable numbers for native and simulated execution.
- **Full-system simulation:** Although several academic papers (e. g., [22, 26]) have proposed fully virtualized HTM implementations, a realistic implementation such as ASF will have quantitative and qualitative limitations on the TM semantics it provides. ASF, for example, aborts ongoing speculative regions whenever there is a timer interrupt, task switch, or page fault. These events are controlled by the OS and potentially have a large impact on performance perceived by code using ASF. To assess this impact, it is therefore necessary to closely model their behavior, which is best done by putting the operating system into the simulation, too.
  PTLsim utilizes Xen's paravirtualized interface [4] to provide a hardware abstraction to the system under simulation. Both applications and the (paravirtualized) OS kernel are simulated in PTLsim's processor core model, making it possible to realistically capture the interplay between ASF and effects caused by the operating system.
- **Detailed, accurate timing model:** Proper OS kernel interaction and identical ISA lay a foundation to generate simulation results that can be compared with results obtained by native execution. Simulation fidelity then largely depends on the accuracy of the simulation models used. For our analysis, we require a detailed timing processor core model that is able to produce results that are similar to those obtained on native AMD Opteron™ processors of families 0Fh (K8 core) and 10h (formerly codenamed "Barcelona").
  Fortunately, PTLsim features a detailed timing model that models an out-of-order core and an associated cache hierarchy in a near-cycle-accurate fashion. We have built on previous tuning attempts [32] and extended the simulator to model the interactions between *multiple* distinct processor cores and memory hierarchies with good tracking of native results [12].

- **Rapid prototyping support:** Detailed simulation models are slower than native execution by several orders of magnitude, because simulating a single cycle usually takes much more than one cycle on the host machine. For our explorative experiments, we cannot pay the extremely high overhead caused by the very detailed RTL-level timing simulators. We believe PTLsim provides a good balance between simulator precision and incurred slowdown, in particular because it allows execution of uninteresting parts of the benchmark runs, such as OS boot and benchmark initialization, at native speed by providing a seamless switchover between native and simulated execution.

  PTLsim's level of modeling and speed of simulation also made it feasible to rapidly prototype and debug different implementations of ASF, while still being able to take an in-depth look at how ASF interacts with features found in current out-of-order microprocessors.

For this work, we have extended PTLsim to support proper multiprocessing by introducing truly separated processor cores and memory hierarchies. We have also implemented a simplified cache-coherence model that accurately captures first-order effects caused by cache coherence [12], but ignores further topology information such as placement of cores on chips or sockets. In addition, we have tuned the characteristics of the simulated core to closely model an AMD Opteron processor.

We have added multiple implementations of ASF to PTLsim and have carefully crafted the interaction between the new ASF functionality and existing mechanisms that enable out-of-order processing. For that, we have modeled additional ordering constraints and fencing semantics for the ASF primitives—we strive for a faithful model of feasible future hardware implementations.

We have currently implemented two of the implementation variants introduced in Section 2.3: LLB-based implementations of varying capacity, and implementations that combine the L1 cache for read-set tracking and an LLB for write-set tracking.

Our simulator has been configured to match the general characteristics of a system based on AMD Opteron processors formerly codenamed "Barcelona" (family 10h), with a three-wide clustered core, out-of-order instruction issuing, and instruction latencies modeled after the AMD Opteron microprocessor [1]. The cache and memory configuration is:

- L1D: 64 KB, virtually indexed, 2-way set associative, 3 cycles load-to-use latency.
- L2: 512 KB, physically indexed, 16-way set associative, 15 cycles load-to-use latency.
- L3: 2 MB, physically indexed, 16-way set associative, 50 cycles load-to-use latency.
- RAM: 210 cycles load-to-use latency.
- D-TLB: 48 L1 entries, fully associative; 512 L2 entries, 4-way set associative.

5. Evaluation

To evaluate ASF, we start by assessing the accuracy of our simulator. That is, we measure the deviation between simulated performance and performance of native execution on a real machine. A close match between simulated and real executions supports our overall approach because it indicates how well the simulator models a realistic processor microarchitecture. Current high-performance x86 microprocessor designs are highly complex and, hence, performance prediction through simulation is nontrivial. To our knowledge, we are the first to try to evaluate this similarity in the TM literature, extending our earlier work [12].

We evaluate ASF using (1) the applications from the STAMP [6] TM benchmark suite[7] and (2) the well-known IntegerSet microbenchmarks. We use the standard STAMP configuration for simulator environments.

IntegerSet runs search, insert, and remove operations on an ordered set of integers, and is implemented either using a linked list, a skip list, a red-black tree, or a hash table. The principles behind these benchmarks resemble the description of the integer-set benchmarks in [11]. Operations are completely random and on random elements. The initial size of a set (i. e., the number of elements in the set) is half the size of the key range from which elements are drawn. No insertion or removal happens if the element is already in or not in the set, respectively. However, we do not have access to the original benchmark code. Hence, some benchmarks (e. g., the red-black tree implementation) could differ slightly. All these programs use several threads and implement synchronization using atomic blocks (i. e., C/C++ transaction statements). We used DTMC to compile the applications and used ASF-TM as the TM library.[8] To reduce impact from the memory allocator, we have selected the allocator with best scalability out of glibc 2.10 standard malloc, glibc 2.10 experimental malloc, and the Hoard memory allocator [5] for the presented results. Runs marked as sequential are single-threaded executions of these programs with no synchronization mechanism in use and no instrumentation added.

Following the performance evaluation, we additionally investigate ASF runtime overheads and the effects of different ASF capacities and ASF's early-release feature.

We use PTLsim-ASF (as described in Section 4) as our simulation testbed. The simulated machine has eight CPU cores, each having a clock speed of 2.2 GHz. Because PTLsim does not yet model limited cross-socket bandwidths, these eight cores behave as if they were located on the same socket, resembling future processors with higher levels of

[7] We exclude the Bayes and Yada applications in our measurements. We have observed nonreproducible behavior for Bayes with several TM implementations, and Yada has extremely long transactions and does not show any scalability with any of the TMs we analyzed.

[8] DTMC is based on LLVM 2.6. Our applications are linked against glibc 2.10.

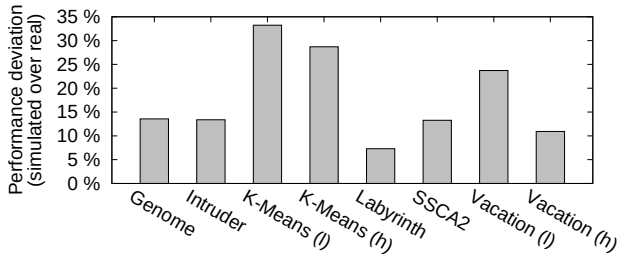

Figure 3. PTLSim accuracy for the runtime of the STAMP benchmarks (no TM, no ASF, one thread) for simulated with respect to native execution.

core integration.[9] We evaluate ASF using four implementations: (1) with an LLB of 8 lines; (2) with an LLB of 256 lines; (3) with L1/LLB of 8 lines; and, (4) with L1/LLB of 256 lines. They are denoted by LLB-8, LLB-256, LLB-8 w/ L1, and LLB-256 w/ L1, respectively. For our STM measurements, we use TinySTM [14] (version tinySTM++ 0.9.9) in write-through mode.

Simulator accuracy. Figure 3 shows the difference in runtimes between execution on a real machine[10] and a simulated execution within PTLsim-ASF, in which we adapted the available parameters of the simulation model to match the characteristics of the native microarchitecture. For five out of the eight STAMP benchmarks, PTLsim-ASF stays within 10–15% of the native performance, which is in line with earlier results for smaller benchmarks [12]. Vacation and K-Means seem to exercise mechanisms in the microarchitecture that perform differently in PTLsim-ASF and in our selected native machine. Clearly, PTLsim cannot model all of the performance relevant microarchitectural subtleties present in native cores, because many of them are not public, highly specific to the revision of the microprocessor, and difficult to reproduce and identify.

One source of the inaccuracies we observed might be a PTLsim quirk: although PTLsim carefully models a TLB and the logic for page-table walks, it only consults them for loads. Stores do not query the TLB and therefore are not delayed by TLB misses, do not update TLB entries, and are not stalled by bandwidth limitations in the page-table walker. The effect on accuracy likely is minor since translations for many stores already reside in the TLB because of a prior load. Nonetheless, we will add a better simulation of stores in a future release of PTLsim-ASF.

Despite these differences, we think that PTLsim models a realistic microarchitecture and captures several novel interactions in current microprocessors. For our main evaluation we conduct all experiments—including the baseline STM runs—inside the simulator to make sure that our results are not affected by the discrepancies.

[9] In a previous study [12], we have analyzed the impact of cross-socket communication for benchmarks of various size.

[10] AMD Opteron processor formerly codenamed "Barcelona," family 10h, 2.2 GHz.

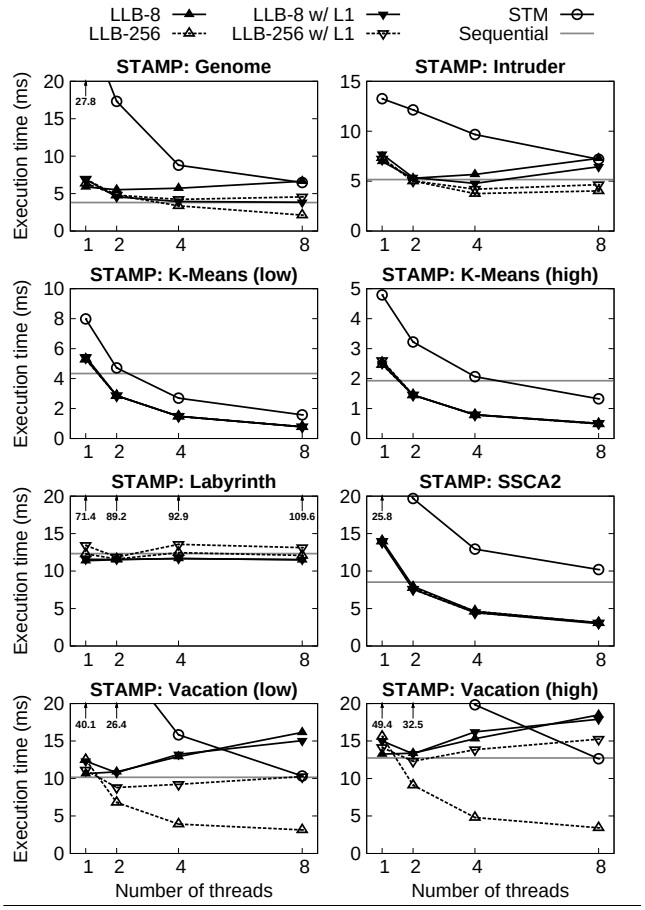

Figure 4. Scalability of applications, with four ASF implementations and varying thread count (execution time; lower is better). The arrows indicate STM values that did not fit into the diagram. The horizontal bars show the execution time for execution of sequential code (without a TM).

ASF performance. Figure 4 presents scalability results for selected applications from the STAMP benchmark suite.[11] We also compare the performance of ASF-based TM to the performance of finely tuned STM (TinySTM) and to serial execution of sequential code (without a TM).

We observe that ASF-based TMs show very good scalability and much better performance than STM for some applications, notably genome, intruder, ssca2, and vacation. Other applications such as labyrinth do not scale well with LLB-8 and LLB-256 because the TM uses serial-irrevocable mode extensively, yet performance is still significantly better than STM. Interestingly, the applications that do not scale well are those with transactions that have large read and write sets (according to Table III in [6]).

For applications with little contention and short transactions, all four ASF variants perform well. For other applications, LLB-256 usually outperforms the other imple-

[11] We added appropriate padding to the entry points of the main data structures to avoid unnecessary contention aborts due to false sharing of cache lines.

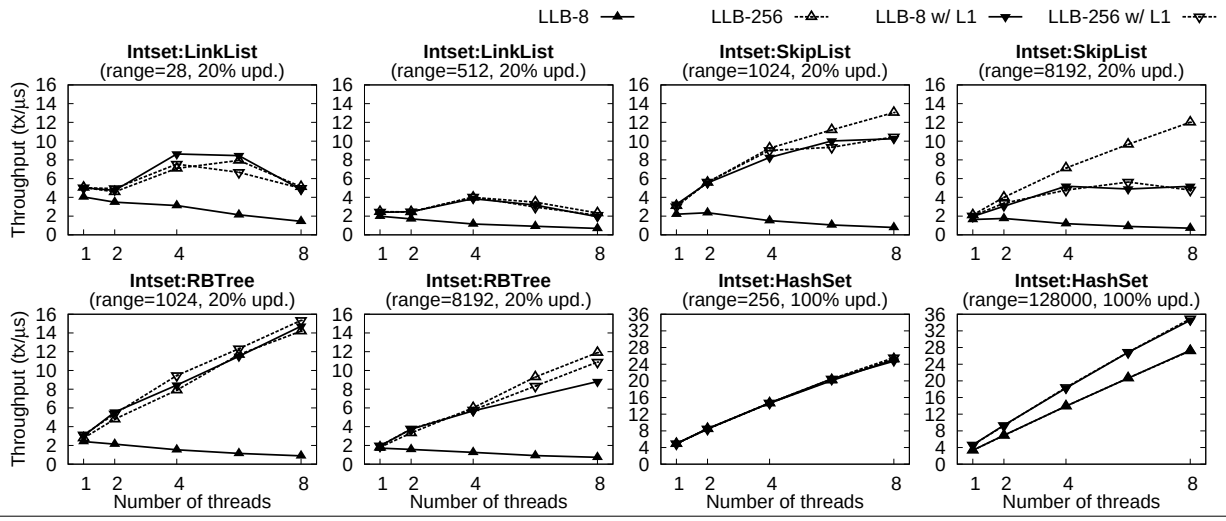

Figure 5. Scalability of IntegerSet with linked list, skip list, red-black tree, and hash set, with four ASF implementations and varying thread count and key range (throughput; higher is better).

mentation variants because LLB-8 suffers from the transaction lengths and L1/LLB is susceptible to cache-associativity limitations. Yet, it is interesting to note, even the LLB-8-based implementation provides benefits for many applications.

To summarize, the ASF-based TMs have a significantly smaller single-thread overhead than the STM and scale well for many benchmarks. The STM-based variants scale as well, but they outperform serial execution only with many threads. In general, the ASF-based TMs outperform the STM by almost an order of magnitude.

Scalability. Figure 5 presents scalability results for the IntegerSet benchmark. We vary the key range between $\{0 \ldots 28\}$ and $\{0 \ldots 128000\}$.

In all IntegerSet variants except the hash-set-based one, the LLB-8 implementation performs poorly because its capacity is insufficient for holding the parts of the data structure that are accessed, leading to constant execution of the software fallback path. This fallback path is serial-irrevocable mode and suffers from contention if used excessively by many threads. The cache-based implementations generally perform equally well, indicating that the write set of all transactions is smaller than 8 cache lines. LLB-256 (without the L1 cache) never performs significantly worse than the cache-based implementations, indicating that the read set always fits into 256 cache lines, and occasionally outperforms them because it is not susceptible to cache-associativity limitations. The performance drop observed for the linked list with more than four threads results from the increased likelihood of conflict in the sequentially traversed list. In general, the hash-set variant performs best and can tolerate the largest key range and the largest update rates because it has the smallest transactional data set and very few conflicts.

ASF abort reasons. Figure 6 provides a breakdown of the abort reasons in the STAMP applications with different ASF implementations. Unsurprisingly, the implementation with the small dedicated buffer (eight-entry LLB) suffers from many capacity aborts for most benchmarks, while the larger dedicated buffer (256-entry LLB) usually has the least capacity aborts. Adding the L1 cache for tracking transactional reads ("+L1") does not always reduce capacity aborts, but actually increases them for several benchmarks. Three reasons contribute to the increase. First, although the L1 cache has a large total capacity, it has limited associativity (two-way set associative) and therefore usable capacity is dependent on address layout. Second, our current read-set-tracking implementation does not modify the cache-line displacement logic. Nonspeculative accesses may displace cache lines used for tracking the read set. Finally, cache lines may be brought into the cache out of order and purely due to speculation of the core. These additional cache lines may further displace lines that track the transaction's read set.

Since displacement of cache lines with transactional data causes capacity aborts, the large number of those is not only caused by actual capacity overflows, but may be caused by disadvantageous transient core behavior. For our current study, we fall back to serial mode to handle capacity aborts, therefore reducing contention aborts for benchmarks with high capacity failures. To leverage the partially transient nature of capacity aborts, one could also retry aborting transactions in ASF and hope for favorable behavior. Furthermore, we will tackle the issue from the hardware side by containing the random effects and ensuring that we meet the architectural minimum capacity. Both aspects are subject of current research.

ASF capacity. Figure 7 presents the scalability in terms of transaction size versus throughput for runs with eight threads. We vary the transaction size (i. e., the number of memory locations accessed) by initially populating the linked list with different amounts of elements. LLB-8 is not sufficient to hold the working set for larger transactions.

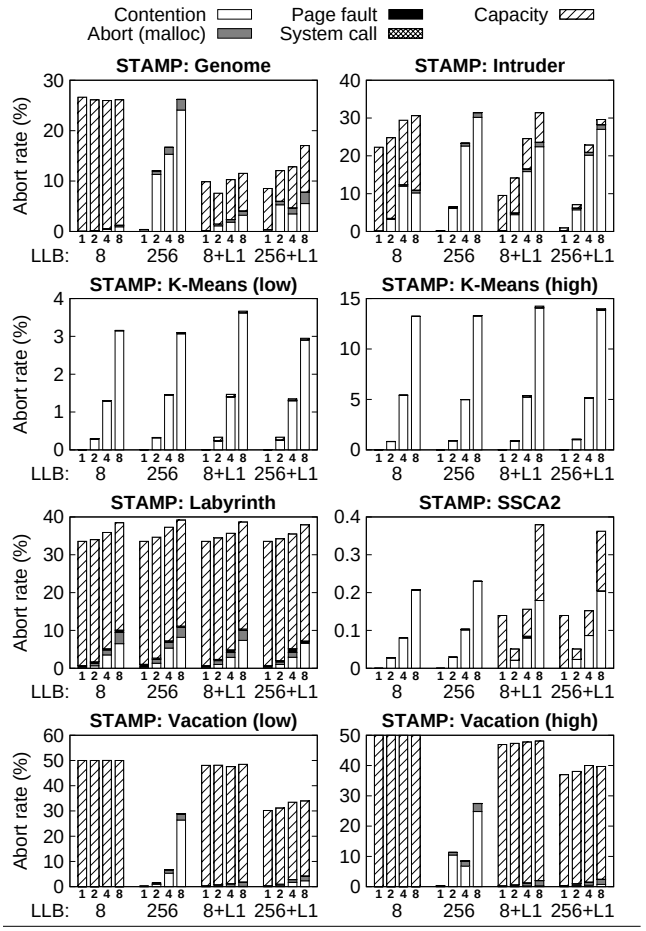

Figure 6. Abort rates of applications, with four ASF implementations and varying thread count. The different patterns identify the cause of aborts.

Transactions have to be executed in software fallback mode for most linked-list transactions with more than eight elements. For the red-black tree, the tree height is most determining for the transaction size. At around 256 elements, almost all transactions run in serial-irrevocable mode for LLB-8.

The overall throughput for the list benchmark decreases with problem size because traversing longer lists increases conflict ratio, work per transaction, and chance for capacity overflow. LLB-256, LLB-8 w/ L1, and LLB-256 w/ L1 behave similarly with this benchmark.

Early release benefits. Figure 8 presents the throughput increase due to the use of early releasing of elements in a transaction's read set. Similar to the well-known hand-over-hand locking technique, we only need to keep the current position in the list in the read set during list traversal. We consider a linked list initially populated with 2^i elements, with $3 \leq i \leq 9$. Using early release makes LLB-8 sufficient because we do not keep all accessed list elements in the read set anymore. Also, throughput increases significantly

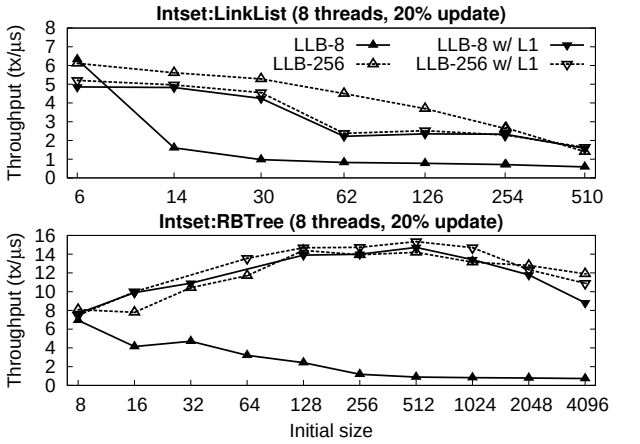

Figure 7. Influence of ASF capacity on throughput for different ASF variants (red-black tree and linked list with 20% update rate with eight threads).

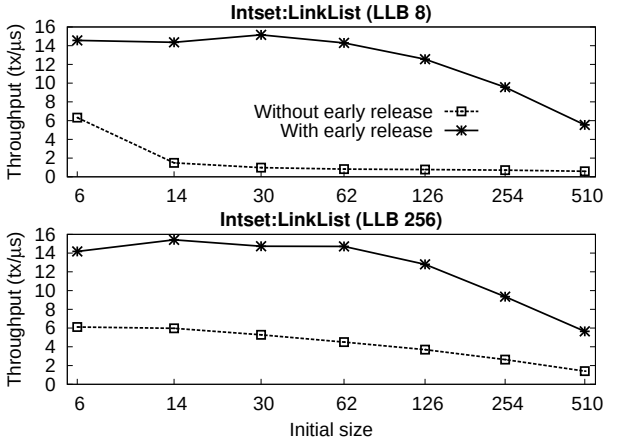

Figure 8. Early release impact: throughput amelioration with linked list (20% update rate, eight threads).

because the chance of conflicts with other transactions decreases.

Although early release has been discussed controversially [30], we think it enables interesting uses cases for expert programmers of lock-free data structures. Our experiments with very limited ASF hardware resources show how early release can increase the applicability of ASF. We acknowledge that early release has complex interactions with the simple TM semantics. Simpler interfaces, such as open nesting [24], and compiler support may simplify this task in the future.

ASF single-thread overheads. To quantify the performance improvement seen with ASF, we have inspected some benchmark runs more closely and broke up the spent cycles into categories. Because adding online timing analysis adds bookkeeping work, interferes with compiler optimization steps, increases cache traffic, and impairs pipeline interaction, we refrained from adding the statistics code into the application or run-time. Instead, we manually annotated the compiled final binaries—marking assembly code line-by-

Application / % updates / size	linked list / 20% / 128			skip list / 20% / 128			red-black tree / 20% / 128			hash set / 100% / 128		
	ASF	STM	Ratio	ASF	STM	Ratio	ASF	STM	Ratio	ASF	STM	Ratio
Non-instr. code	0	0	–	0	0	–	0	0	–	9738	0	0.00
Instr. app. code	1368105	1747385	1.28	1107561	1793351	1.62	2039471	281328	0.13	78822	87368	1.11
Abort/restart	0	0	–	0	0	–	0	0	–	426147	0	0.00
Tx load/store	1029659	31024930	30.13	652817	10073146	15.43	233246	7623913	32.69	533696	5013248	9.39
Tx start/commit	1322509	1087201	0.82	1276152	1176545	0.92	1306687	1033154	0.79	1263550	1316656	1.04

Table 1. Single-thread breakdown of cycles spent inside transactions for ASF-TM (with LLB-256) and TinySTM.

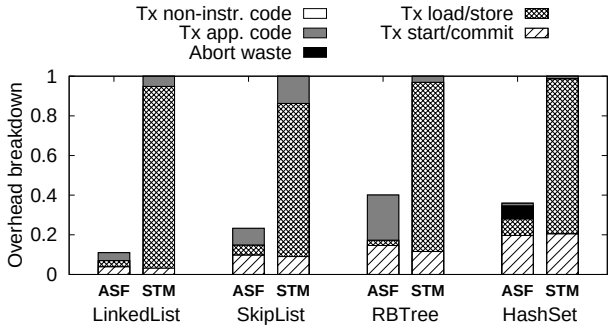

Figure 9. Single-thread overhead details for ASF-TM (with LLB-256) and TinySTM. All values normalized to the STM results of the respective benchmark.

line with one of the categories "TX entry/exit," "TX load/store," "TX abort," and "application"—and extended our simulator to produce a timed trace of the execution. We then produced the cycle breakdown by offline analysis and aggregation of the traces, without any interference with the benchmarks execution.

Figure 9 and Table 1 present the details of the composition of the overhead imposed by the TM stack based on ASF or on STM. The results are for single-threaded runs of the IntegerSet benchmark on the LLB-256 implementation. Because there is only one thread, there are no aborts caused by memory contention. All aborts reported for the hash-set variant occur because of page faults, which require OS-kernel intervention and therefore abort the ASF speculative regions.

The overhead of starting and committing transactions is similar for ASF and STM in single-thread executions, largely due to the additional code that is run for entering a transaction. As described in Section 3.2, we had to add code to the ASF implementation that provides the semantics of the ABI on top of the `SPECULATE` instruction. For small transactions, this cost can be the dominant overhead in comparison to the uninstrumented code. Looking at ways to integrate the ASF primitives more directly, and thus with less overhead, is one of our future topics, because it requires more extensive transformations in LLVM.

Although transactional loads and stores are much more costly in general in an STM, we were surprised by the difference in improvement for different benchmarks. If we compare the red-black tree and the hash set, we find that there is almost a factor of 33× speed-up for transactional loads and stores for the tree, and only 9× for the hash-set. On closer inspection we found that this can be attributed to cache effects: the hash set has many cache misses, because its data access pattern is mostly random and all accesses update the set, which in total is larger than the first and second level caches (2^{17} buckets, with 16 bytes/bucket). With out-of-order execution, a large part of the STM's constant additional computation and memory traffic overhead can be effectively interleaved with the cache misses and in general has less impact on the incurred relative slowdown.

The additional aborts due to semantic limitations of ASF (see Section 3.3) have negligible performance impact for our single-threaded measurements.

6. Related work

The first hardware TM design was proposed by Herlihy and Moss [16]. It is an academic proposal that does not address the capacity constraints of modern hardware. A separate transactional data cache is accessed in parallel to the conventional data cache. Introducing such a parallel data cache would be intrusive to the implementation of the main load-store path. This would require massive modifications that would make this mechanism impractical to add to current microprocessors. By contrast, ASF can be implemented without changes to the cache hierarchy.

Shriraman et al. [29] propose two hardware mechanisms intended to accelerate an STM system: alert-on-update and programmable data isolation. The latter mechanism, which is used for data versioning, relies on heavy modifications to the processor's cache-coherence protocol: the proposed TMESI protocol extends the standard MESI protocol (four states, 14 state transitions) with another five states and 30 state transitions. We regard this explosion of hardware complexity as incompatible with our goal of being viable for inclusion in a high-volume commercial microprocessor.

Rajwar et al. [26] propose a virtualized transactional memory (VTM) to hide platform-specific resource limitations. This approach increases the hardware complexity unnecessarily: we strongly believe virtualization is better handled in software.

Several other academic proposals for hardware TM have been published more recently. To keep architectural extensions modest, proposals primarily either restrain the size of supported hardware transactions (e. g., HyTm [9, 19], PhTM [21]), or limit the offered expressiveness (e. g., LogTM-SE [31], SigTM [23]). Each of these hardware approaches is accompanied by software that works around the limitations and provides the interface and features of STM: flexibility, expressiveness, and large transaction sizes.

Our work differs from these approaches in several respects. First, ASF requires no changes to the cache-coherence protocol and no additional CPU data structures for bookkeeping, such as memory-address signatures or logs. Second, ASF does not depend on a runtime system and can be used in all system components including the OS kernel. Finally, we evaluated ASF using two hardware-inspired implementations for an out-of-order x86 core simulator, giving us high confidence that ASF can be implemented in a high-volume commercial microprocessor.

Intel's HASTM [28] is an industry proposal for accelerating transactions executed entirely in software. It consists of ISA extensions and hardware mechanisms that together improve STM performance. The proposal allows for a reasonable, low-cost hardware implementation and provides performance comparable to HTM for some types of workloads. However, because the hardware supports read-set monitoring only, it has fewer application scenarios than HTM. For instance, it cannot support most lock-free algorithms.

Sun's Rock processor [11] is an architectural proposal for TM that was actually implemented in hardware. It it based on the sensible approach that hardware should only provide limited support for common cases and advanced functions must be provided in software. Early experiences with this processor have shown encouraging results but also revealed some hardware limitations that severely limit performance. Notably, unlike ASF, TLB misses abort transactions. Rock also does not support selective annotation, as described in Section 2. Finally, Rock does not provide any liveness guarantee, so lock-free algorithms cannot rely on forward progress and have to provide a conventional second code path. By contrast, ASF does ensure forward progress when protecting not more than four memory lines at least in the absence of contention.

Azul Systems [8] has developed multicore processors with HTM mechanisms built in. These mechanisms are principally used for lock elision in Java to accelerate locking. The solution appears to be tightly integrated with the proprietary software stack, so not a general-purpose solution. It also does not support selective annotation like ASF.

Diestelhorst and Hohmuth [12] described an earlier version of ASF, dubbed ASF1, and evaluated it for accelerating an STM library. The main difference between ASF1 and the current revision, ASF2, is that ASF1 did not allow dynamic expansion of the set of protected memory locations once a transaction had started the atomic phase in which it could speculatively write to protected memory locations. As a consequence of this restriction, the ASF1-enabled STM system used ASF1 only for read-set monitoring (because the read set could be expanded dynamically) and resorted to purely software-based versioning. The resulting hybrid STM system cannot be compared directly to ASF-TM because it did not require serialization in case of capacity overruns. In general, it performed slightly better than TinySTM for the red-black-tree and linked-list microbenchmarks presented in Section 5 ($\approx$10 % performance improvement with an LLB-8 configuration).

7. Conclusion

In this paper, we have presented a system that permits an efficient execution of transactional programs. It consists of a full system stack comprised of ASF, an experimental AMD64 architecture extension for parallel programming; three proposals for ASF hardware implementations; a compiler, DTMC, for transactional programs; and, a runtime, ASF-TM. Our evaluation indicates that this system improves performance compared to previous software-only solutions by a significant margin, and provides good scalability with most workloads.

We also presented PTLsim-ASF, a version of the full-system out-of-order-core AMD64 simulator PTLsim that we enhanced with a faithful simulation of ASF. Our simulator closely tracks the native execution performance of current AMD CPUs, giving us confidence in our measurement results.

Unlike many previous hardware-acceleration proposals for TM systems, ASF has been developed in the framework of constraints that apply to the development of modern high-volume microprocessors. We hope to help other researchers get a better understanding of how constrained this environment is, and what realistically can be expected in terms of TM acceleration from future CPU products. Nonetheless, ASF does provide a number of novel features, including selective annotation and an architecturally ensured minimum transaction capacity.

Our transactional-memory compiler, DTMC, directly targets ASF via ASF-TM, our TM runtime. We have demonstrated that, for the workloads we analyzed, no sophisticated STM system is needed to maintain good performance in most cases. A serializing software fallback mode plus a few optimizations aimed at requiring fewer ASF aborts were sufficient.

We plan to make DTMC, ASF-TM, and PTLsim-ASF publicly available to give early adopters a chance to try out ASF.

Acknowledgments

We are very grateful to Richard Henderson and Aldy Hernandez of Red Hat for working on and sharing `gcc-tm`.

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement N^{o} 216852.

References

[1] *Software Optimization Guide for AMD Family 10h Processors*. Advanced Micro Devices, Inc., 3.05 edition, Jan. 2007.

[2] *Advanced Synchronization Facility - Proposed Architectural Specification*. Advanced Micro Devices, Inc., 2.1 edition, Mar. 2009.

[3] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *Readings in computer architecture*, 2000.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM symposium on Operating systems principles (SOSP)*, Boston Landing, NY, USA, 2003.

[5] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the 9th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, 2000.

[6] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multiprocessing. In *Proceedings of The IEEE International Symposium on Workload Characterization (IISWC)*, Seattle, WA, USA, Sept. 2008.

[7] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Commun. ACM*, 51(11), 2008.

[8] C. Click. Azul's experiences with hardware transactional memory. In *HP Labs - Bay Area Workshop on Transactional Memory*, Jan. 2009.

[9] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, San Jose, CA, USA, 2006.

[10] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, Stockholm, Sweden, 2006.

[11] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, Washington, DC, USA, 2009.

[12] S. Diestelhorst and M. Hohmuth. Hardware acceleration for lock-free data structures and software-transactional memory. In *Proceedings of the Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods (EPHAM)*, Boston, MA, USA, Apr. 2008.

[13] P. Felber, C. Fetzer, U. Müller, T. Riegel, M. Süßkraut, and H. Sturzrehm. Transactifying applications using an open compiler framework. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, Portland, OR, USA, Aug. 2007.

[14] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Salt Lake City, UT, USA, Feb. 2008.

[15] M. Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Seattle, WA, USA, 1990.

[16] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, San Diego, CA, USA, May 1993.

[17] Intel. *Draft Specification of Transactional Language Constructs for C++*. Intel, IBM, Sun, 1.0 edition, Aug. 2009.

[18] Intel. *Intel Transactional Memory Compiler and Runtime Application Binary Interface*. Intel, 1.0.1 edition, Nov. 2008.

[19] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proceedings of the 11th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, New York City, NY, USA, Mar. 2006.

[20] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, Palo Alto, CA, USA, Mar. 2004.

[21] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, Portland, OR, USA, Aug. 2007.

[22] S. Lie. Hardware support for unbounded transactional memory. Master's thesis, May 2004. Massachusetts Institute of Technology.

[23] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. *SIGARCH Comput. Archit. News*, 35(2), 2007.

[24] J. E. B. Moss and A. L. Hosking. Nested transactional memory: model and architecture sketches. *Sci. Comput. Program.*, 63(2), 2006. ISSN 0167-6423.

[25] R. Rajwar and J. R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Proceedings of the 34th ACM/IEEE International Symposium on Microarchitecture (MICRO)*, Austin, TX, USA, Dec. 2001.

[26] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, Washington, DC, USA, 2005.

[27] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, New York, NY, USA, Mar. 2006.

[28] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Washington, DC, USA, 2006.

[29] A. Shriraman, M. F. Spear, H. Hossain, V. Marathe, S. Dwarkadas, and M. L. Scott. An integrated hardware-software approach to flexible transactional memory. In *Proceedings of the 34th annual international symposium on Computer architecture (ISCA)*, San Diego, CA, USA, 2007.

[30] T. Skare and C. Kozyrakis. Early release: Friend or foe? In *Workshop on Transactional Memory Workloads*. Jun 2006.

[31] L. Yen, J. Bobba, M. M. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Phoenix, AR, USA, 2007.

[32] M. T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2007.